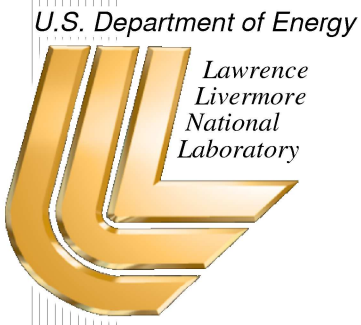
UCRL-MA-153588

Merlot Design



June 13, 2003 - Version 0.3

Approved for public release; further dissemination unlimited

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

UCRL-MA-153588

Merlot Design

Sean Ahern – LLNL

Merlot Design

Version history:

2/14/03: Version 0.1 (Sean Ahern – LLNL)
6/10/03: Version 0.2 (Sean Ahern – LLNL)
6/13/03: Version 0.3 (Sean Ahern – LLNL)

1. Abstract

We describe Merlot, a system for delivery of digital imagery over high speed networks. We describe various use cases, the client/server interaction, and the image and network codecs. We also describe some possible applications using Merlot and future work.

2. Motivation

LLNL has many large visualization systems (SGIs, various clusters, etc.) that are used for large-scale visualization work. In general these systems are used for visualization in one of three ways:

1. An analog video signal is routed through a video switch to individual offices by way of a fiber-optic connection.
2. Analog video signals are routed through a video switch to the projectors/panels making up a tiled display such as a powerwall.
3. As a large sort-last software-based rendering system, usually using Mesa.

While these methods have been successful, they have a number of problems.

For the first two analog-delivery methods, there is a mismatch between the number of video outputs of the visualization system(s) and the number of users who desire high-end visualization capabilities. While a video switch can help move resources around, this can become rather cumbersome to schedule and control.

The analog video signal methods also have the problem of distance. The video signals can only be extended so far until signal degradation makes them unusable. As more visualization systems come on line, a hardware-accelerated solution other than fiber-optics must be found.

The Mesa solution does not tap the hardware-acceleration features of the machine, and is thus not a reasonable general solution for large-scale visualization.

LLNL is currently in the planning stages of ASCI purple deployment. It is expected that we will wish to deliver imagery at high speeds from Purple and its accompanying machine to users' offices. Because of the limitations mentioned above, we have a need to find an alternate image delivery system than we currently have deployed.

We investigated a system for redirected OpenGL delivery for use in predicting the network requirements for future deployments. In the process of that analysis, it became apparent that abstracting image compression and network transport would be

very valuable. This document discusses the architecture of the image delivery framework. We briefly discuss our application for OpenGL redirection in section 5.1.

3. Use cases

In considering a new design for digital delivery of imagery, we considered two major use cases:

1. Quantitative visualization work in an office. A user is running a quantitative visualization application like EnSight, VisIt, or Paraview. They want very accurate imagery, and they are willing to wait some amount of time as long as the image is guaranteed to be correct. Since the user is actively interacting with the imagery, latency must be minimized.

2. Playing an animation. In this case, imagery must be delivered very quickly to a destination. Since the acuity of the human eye decreases when viewing imagery in motion, detail is not as important. This use case values throughput greater than accuracy. High latency is acceptable in this case.

The machines on which the sender and receiver are located may have more than one network interface between them. Due to the constraints of multi-person usage of the machines, one or more of the network interfaces may be filled with other traffic. Merlot could conceivably be asked to detect this and change its communication paths to most efficiently take advantage of the available network bandwidth.

Discussions with external collaborators have revealed another possible use of Merlot that must be considered. In many circumstances, there may be more than one source of imagery and many destinations for the same. While we have primarily considered the point-to-point case, there may be design considerations that can help facilitate the fan-in and fan-out nature of cluster-based image delivery to multiple destinations. We discuss these possibilities in section 6.2.

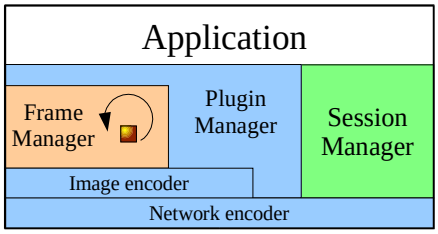
4. Design

4.1. Overview

Fundamentally, Merlot is an infrastructure for encoding and delivering imagery from point to point on a network. The work of image encoding/decoding and the work of network delivery are performed by interchangeable plugins, similar to Photoshop or Netscape plugins. Merlot is intended to be a delivery vehicle for research into image encoding and network delivery. As such, we hope that existing work in this area can be incorporated into Merlot with little modification.

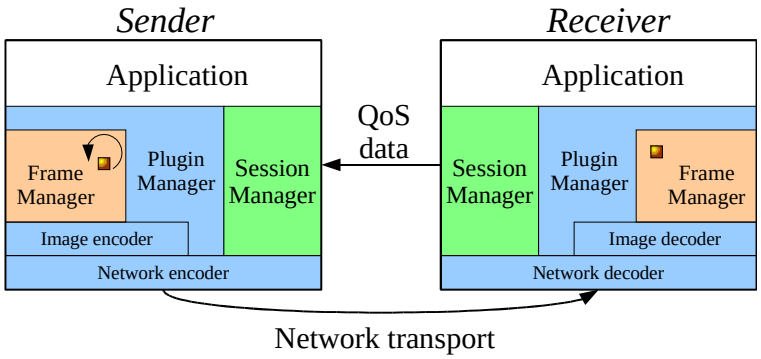
Merlot is a client/server architecture, consisting of a client API and a server API, each linked into an application. The two sides are connected by a network connection, with encoded imagery and quality-of-service data being delivered between them. In Merlot terminology, the two sides are called the “sender” and the “receiver”. Each endpoint is called a “node”.

A node consists of several parts, as seen in the figure to the right.



When the application initializes Merlot, the the plugin manager begin its work. The plugin manager handles the creation and destruction of the image and network codecs, as well creation and management of the frame manager. When an image is read to be sent to a receiver, the application hands it to the frame manager, which keeps a circular buffer of previous frames. It then informs the image encoder that a new frame is ready. The image encoder encodes the frame, and hands it to the network encoder for transmission.

The reverse process occurs at the receiver end of the network. Information about when the frame was displayed at the receiver is recorded by the session manager. This information is communicated back to the sender's session manager so that the codecs may tune their performance. The application may even decide to swap out the codecs for other ones to tune performance. A full diagram of this architecture may be seen in the figure above.



4.2. Time sequence of operations

The details of execution are best considered through a description of the time sequence of how Merlot operates.

(This is a fairly high-level description, though it gets into some ideas for implementation. Several API calls are described here. We have not given their prototypes, since we haven't fully fleshed out all of the information in them; this is a work in progress.)

Both the sender and receiver processes are started. (See section 6.4 for launching concerns.) They each call `MerlotInit` to initialize the system. Two options are set through the `MerlotInit` call. First, which role this Merlot node will be playing, sender or receiver. If the call is made by the sender, the location of the receiver on the network is also given. If the call is made by the receiver, the TCP port to listen on is specified. Merlot then loads its configuration file, which defines what image and network plugins are to be used.

The sender opens a TCP connection to the receiver and they handshake, sharing compatibility information such as byte encoding, desired plugins, etc. This socket is not expected to have any particular performance characteristics, as it is only used

for control data and quality of service information. The communication of the frame data is the responsibility of the network codec.

After the sender and receiver have done a handshake, `MerlotInit` sets up the session manager, the frame manager, and the plugin manager. The plugin manager starts separate threads for the image codec and the network encoder. We use threads for each of these portions so that computation may be overlapped when possible. The plugin manager loads the shared library plugins, calling `MerlotLoad` in each of them to determine information characterizing the plugin.

Note that, to guarantee compatibility, the image encoder and the image decoder must match, as well as the network encoder and decoder. The three sender threads are dedicated to the application, the image encoder, and the network encoder, respectively. The two encoder threads block, waiting for the frame manager to signal that a frame is ready.

`MerlotInit` on the receiver spawns similar threads, one for network decoding and one for image decoding. The two threads block, waiting for image data from the sender.

The sender application generates frames, giving them to Merlot through the `MerlotSendFrame` call. The frame manager adds the frame to its circular buffer of frames and signals the plugin manager that work may begin on the frame. The plugin manager tells the image encoder to begin its work. It then returns control to the application. In the meantime, the image encoder starts work on the frame in the circular buffer. Since the image encoder may take a significant amount of time to encode a frame in comparison to the speed that frames are generated, the image encoder has access to the entire circular buffer, allowing the image encoder to drop frames as needed or perform lookahead. Once the image encoder finishes encoding a frame or a partial frame, it passes the data back up to the plugin manager for delivery. The plugin manager calls the network encoder thread to communicate the encoded frame data to the receiver. The network encoder contacts the network decoder across the network and communicates the frame data. The network encoder informs the session manager that it has communicated the frame.

On the other side of the network, the receiver's network decoder decodes the data and delivers it to the plugin manager. The plugin manager hands the data to the image decoder. The image decoder uses the encoded data to create a final frame. This frame is given to the frame manager for availability to the receiver application. The timestamp of this delivery is recorded by the session manager, which communicates it across the control socket back to the sender's session manager. The receiver application uses an API call to retrieve the latest computed image. We haven't decided yet whether the API for the receiving application should be a blocking or a non-blocking API.

4.3. Image codec details

Image codecs can take two forms: progressive and non-progressive. A non-progressive codec is one that encodes an entire frame into a single message. This

data is delivered with one call to the network encoder, and the entire frame's data is sent across the network to be reconstructed at the receiver.

Alternately, a progressive codec encodes a single frame into multiple data messages, communicating each message separately across the network. One can imagine a case where the low frequency portions of an image are identified and communicated to the receiver, with higher-frequency portions communicated with additional messages as time progresses. If no additional frame is provided by the application, thus interrupting the progressive image codec, the entire frame would eventually be communicated to the receiver. If a new frame is provided by the application, the progressivity would be halted, and computation started on a new image. Because of this, it is important for progressive codecs to continue to check with the frame manager for the availability of new image data.

On the receiver side, each phase of a progressive codec generates an entire image. This allows the generation of images with increasing detail, yet still providing high frame rates.

In neither the progressive nor non-progressive cases are the codecs required to reproduce the input image exactly. They may do so, but lossy compression has many advantages. The only stipulation is that the output of an image decoder must generate a full frame at every stage.

4.4. Network codecs

The image codecs have no knowledge of the network infrastructure. The image encoder sends its encoded data to the plugin manager. The plugin manager sends this information to the network encoder for communication. All duties of creating, using, and tearing down the network connection are the responsibility of the network encoder. Initially, the network encoder and decoder are given information about where on the network its counterpart resides, as well as a chance to handshake across a TCP connection. Using this location information, the sender and receiver portions of the network codec create one or more connections at initialization time. Note that TCP is only used for handshaking. No requirement for TCP is made for actual data communication.

4.5. Merged codecs

Note that the image codec and the network codec operate independently of each other. Thus, a merged codec that incorporates both aspects of image encoding and network delivery may be put into place. In this form, once the codec finishes encoding an image, it may immediately take action to move it across a network. This may make sense for hardware-accelerated infrastructures such as Sepia, which perform both operations at once.

4.6. Plugin manager

The plugin manager serves as a thin "service" layer between the image codec and the network codec. It also handles the instantiation and destruction of the codecs. Since all plugins answer to the same API, we envision the plugin manager

providing the ability to swap out image and network codecs at any point during runtime, so as to address changing needs.

For example, if the user changes actions, such that instead of viewing a dataset in detail, he starts to play a movie of a simulation, the plugin manager might be instructed to swap out the existing image codec and instead load a high-latency, lossy image codec.

The plugin manager also handles the creation and management of the frame manager, so that the specific instance can be tuned for the needs of the codecs. (For instance, an image codec that does not perform frame dropping or lookahead does not need a circular buffer.)

4.7. Configuration information

We anticipate that the user will want to modify the behavior of the codecs at initialization time and during runtime. A communication pathway from the application, down through the plugin manager, and into the image and network codecs, is needed for this configuration information. We envision a generalized preference mechanism based on XML would provide the carrier for this info.

4.8. Session manager feedback information

The information that the session manager gathers is important for ensuring a steady image flow. A family of protocols currently exist for streaming multimedia data (RFC1889, RFC2326). We are currently investigating in what form we will deliver this capability.

5. Applications

Using this design, we see several immediate applications which could benefit.

5.1. OpenGL redirection application

The initial use of the Merlot library will be in the deployment of an application for OpenGL redirection, MIDAS (Merlot Image Delivery Application Server). The sender application is an unmodified X11/OpenGL application that has been "tricked" at runtime into loading a custom library that understands Merlot.

This custom library contacts a MIDAS receiver across the network through a TCP connection and shares information about display specifications, window locations, etc. The OpenGL stream is redirected to a high-end display for rendering. When a frame is finished (upon receipt of a `glSwapBuffers` call, for instance), the frame is read into memory through a `glReadPixels` call and is given to Merlot for delivery. The receiver receives the frame from Merlot and blits it to the original application window.

5.2. Chromium

The readback SPU of Chromium is an ideal image source. We will create a sub-SPU of readback that contains the Merlot sender library.

It's not yet clear what would be on the other side of the network. It could be another SPU which receives the imagery and turns it into a `glDrawPixels` call to be further processed through Chromium. But other scenarios are possible.

5.3. PICA

Merlot could be considered a single communication pathway through the PICA system. This avenue has yet to be explored.

5.4. VNC

Merlot could conceivably be used as an encoder for the VNC system. While VNC is fundamentally a "pull" system, while Merlot has been designed in a "push" fashion, we would like to explore this possibility.

6. Future work and unsolved issues

6.1. Full-screen hardware encoders/decoders

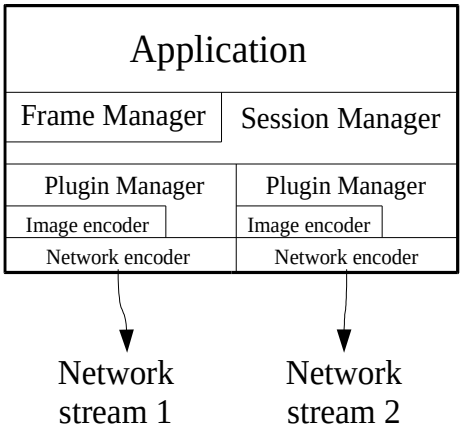
Merlot was originally intended for use in MIDAS. As such, it has been designed with the idea of processing imagery from and to an OpenGL window.

However, the Merlot library, separated from MIDAS, has no such dependence on windows. An image source could easily be an entire screen, captured by a video capture board or through other mechanisms. Much research is being done in this area, especially at Sandia National Laboratories. While we at LLNL are initially going to focus on the "in-window" portion of Merlot, we would like to explore the full-frame perspective.

6.2. Multiple fan-in/fan-out

Some user environments require a final image to be generated by multiple sources, such as a cluster in a sort-first configuration. This final image may also need to be displayed on multiple destinations, as in a tiled-display or a collaborative system. Merlot was originally designed only to address the point-to-point problem. However, we do not wish to prevent the ability to use many Merlot streams to create a system that provides multiple inputs and outputs.

A possible way of providing multiple fan-outs is to have multiple image/network encoders in a single node. The session manager would load multiple plugin managers, each handling one set of encoders. This session manager would be aware of the multiple destinations, and would be able to respond to performance requests by the individual streams. A diagram of such a node might be as seen at the right.



Similarly, a receiver node may have multiple network and image decoders.

However, while the encoding and decoding of images can certainly happen through a mechanism such as this, issues such as varying performance of the links, different desired frame sizes, and overlapping imagery still have to be worked out. This work is intended to be done after initial deployment of applications using the point-to-point version of Merlot has occurred. We include this description here as a seed for discussion.

6.3. Synchronized image streams

It might advantageous to use Merlot in parallel, communicating each stream through a Merlot sender and receiver so as to present an image or set of imagery on a tiled display. The synchronization of the streams would be needed. This has not yet been explored.

6.4. Launching

Some network architectures (MPI, for instance), require that all communication participants be launched at the same time. Or they may require that all communicating processes exist when the network resources are requested. Our initial work will be TCP/IP-based, where this is not a requirement. But with varying network architectures in mind, no constraints are placed on the launch order of the sender and receiver processes, as long as they both participate in the initial handshake.

For ease of use and user transparency, one can imagine environments where the receiver side of Merlot is automatically launched by the sender through use of ssh or other mechanisms. While we are not addressing this need during initial development, we anticipate that we will need to provide an simple launching mechanism for final deployment.